BLOCKMAKER LIVE

Guía de ejercicios

Módulo 7

Paso 0. Crear el contrato y descripción general 5
Paso 1. Introducir cabecera contracto 5
Paso 2. Declarar “contract” Mercado 5
Paso 3. Crear variable propietario 5
Paso 4. Crear variable Producto 5
Paso 5. Crear mapping 6
Paso 6. Crear contador de productos 6
Paso 7. Crear evento ProductoAgregado 6
Paso 8. Crear evento ProductoComprado 6
Paso 9. Crear modificador soloPropietario 6
Paso 10. Crear modificador productoExistente 7
Paso 11. Crear constructor 7
Paso 12. Implementar función agregarProducto 7
Paso 13. Implementar función comprarProducto 8
Paso 14. Implementar función obtenerProducto 9
(EXTRA - No realizar). Declarar función “receive” 9

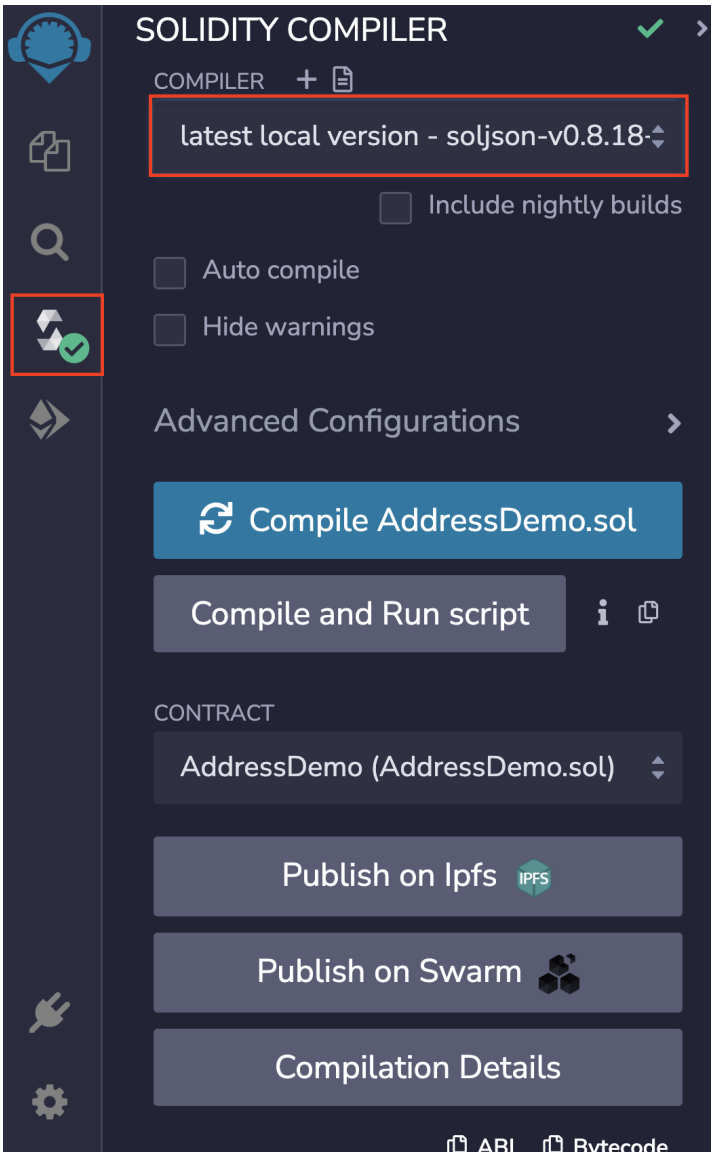
Recordatorios

1.A continuación, os dejamos un enlace a modo de bibliografía y de utilidad para consultar cualquier duda en materia de Solidity (si lo necesitáis, usar la función de traducir página al español):

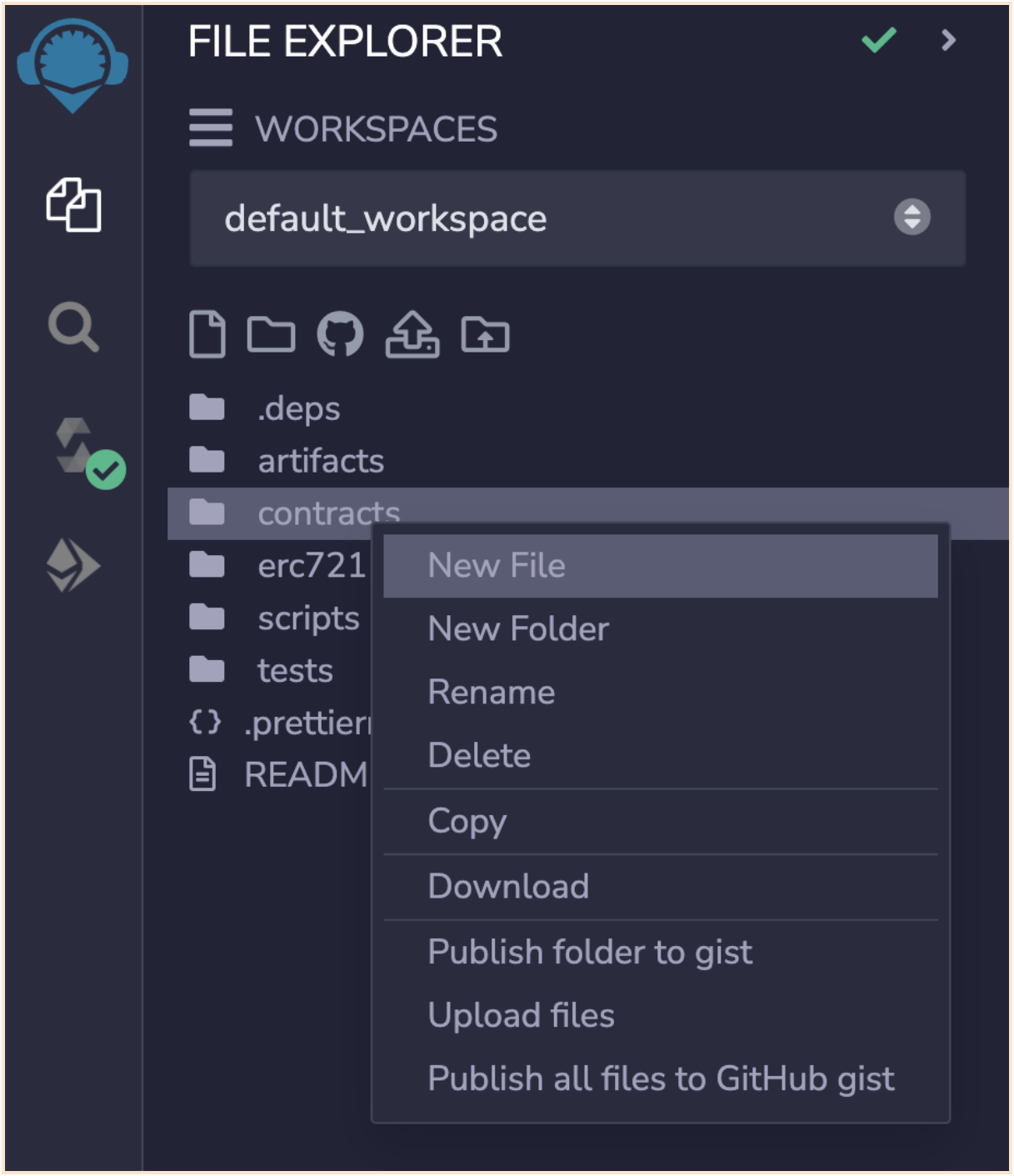
- https://docs.soliditylang.org/en/v0.8.21/grammar.html

2.Para realizar estos ejercicios vamos a utilizar el compilador online de Remix http://remix.ethereum.org/

RECUERDA 1: Desde la página http://remix.ethereum.org/ vamos a irnos a la tercera sección de la izquierda y vamos a cambiar la versión del compilador. Para ello, seleccionaremos cualquiera superior a la 0.8.0:



RECUERDA 2: Para crear un nuevo contrato, nos iremos a la primera sección de la izquierda y colocaremos en la carpeta “contracts” -> “New File” y le pondremos una nombre con la extensión “.sol.



Paso 0. Crear el contrato y descripción general

Crea un nuevo contrato Solidity con el nombre "**Mercado**."

Descripción del contrato: Este contrato representa un mercado básico donde el propietario puede agregar productos y los usuarios pueden comprarlos. El contrato tiene eventos (ProductoAgregado y ProductoComprado) para registrar la adición de productos y las compras.

Paso 1. Introducir cabecera contracto

Copiar al inicio del contrato inteligente.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;
```

Paso 2. Declarar “contract” Mercado

Creamos el **contract Mercado** (recuerda que se declara como si fuera una clase y se le añaden las llaves para introducir dentro todo el contenido).

Paso 3. Crear variable propietario

Define una **variable de estado llamada "propietario"** de tipo **address** y haz que sea **pública (public)**. Esta variable almacenará la dirección del propietario del contrato.

Paso 4. Crear variable Producto

Define una **variable de estado llamada "Producto"** de tipo **struct**.
Esta variable contendrá las siguientes propiedades:

- id (entero)
- nombre (cadena)
- precio (entero)
- cantidad (entero)
- vendedor (address)

Esta variable almacenará los datos específicos de cada tipología de producto.

Paso 5. Crear mapping

Crea un mapeo llamado "**productos**" que mapea **el identificador del producto** (de tipo **entero**) con su **Producto** (de tipo **struct**) correspondiente. Este mapeo debe ser **privado (private)**.

Paso 6. Crear contador de productos

Crea un **entero** llamado "**contadorProductos**" con visibilidad **pública** que almacenará el número de productos que están presentes en la aplicación.

Paso 7. Crear evento ProductoAgregado

Crea un **evento (event)** llamado "**ProductoAgregado**" con los siguientes datos:

- idProducto (entero)
- precio (entero)
- cantidad (entero)
- vendedor (address)

Este evento almacenará el historial de todos los productos que se han agregado al mercado y el detalle del precio, cantidad y vendedor.

Paso 8. Crear evento ProductoComprado

Crea un **evento (event)** llamado "**ProductoComprado**" con los siguientes datos:

- idProducto (entero)
- precio (entero)
- cantidad (entero)
- comprador (address)

Este evento almacenará el historial de todos los productos que se han comprado en el mercado y el detalle del precio, cantidad y comprado.

Paso 9. Crear modificador soloPropietario

Crea un **modificador (modifier)** llamado "**soloPropietario**" el cual no requiere ningún tipo de variable de entrada y comprobará que el **emisor de la petición (msg.sender) es el propietario**, en caso contrario, fallará.

Paso 10. Crear modificador productoExistente

Crea un **modificador (modifier)** llamado "**productoExistente**" el cual require como dato de entrada:

- idProducto (entero).

Este modificador comprobará:

1. Que el **idProducto es distinto de 0**.
2. Que el **idProducto es menor o igual que el contadorProductos** (ya que el contador productos nos dirá cuál es el identificador más alto del último producto que se ha agregado al mercado).

Paso 11. Crear constructor

Crea una **función constructora (constructor) que establezca el valor de "propietario" como la dirección que despliega el contrato** y que inicialice **contradorProductos a 0**.

Paso 12. Implementar función agregarProducto

Implementar la función "**agregarProducto**" **pública que agregará un nuevo producto al mercado.**

***Parámetros de entrada**:*

- Variable **nombre** de tipo **cadena**.
- Variable **precio** de tipo **entero**.
- Variable **cantidad** de tipo **entero**.

***Modificador de función**:*

- soloPropietario.

***Descripción**:*

- Esta función **incrementará en 1 el contadorProductos** (ya que llevará la contabilidad de todos los productos que se van dando de alta y les asignará ese identificador de manera secuencial)**.**
- Añadirá al **mapping el nuevo identificador de producto que será el propio contadorProductos** y lo **igualará al nuevo producto de tipo Producto (struct)**, pasándole todos los datos de entradas que nos han llegado junto al propio **contadorProductos (como id)** y el **msg.sender (como vendedor)**.
- Emitimos el **evento "ProductoAgregador" pasándole todos los datos que requiere dicho evento** (pista, los datos erán los mismos que le hemos pasado al nuevo Producto struct).

Paso 13. Implementar función comprarProducto

Implementar la función ”**comprarProducto**” **pública** de tipo **payable que permitirá comprar un del mercado con ether.**

***Parámetros de entrada**:*

- Variable **idProducto** de tipo **entero**.

***Modificador de función**:*

- productoExistente(idProducto).

***Descripción**:*

- Esta función **comprobará (require) si hay stock (cantidad) del producto en cuestión (buscar en el mapping el idProducto y obtener el campo “cantidad” del struct).**
- Tras esto, se comprobará que **el ether que ha enviado el emisor (msg.value) es superior o igual al precio del producto** que quiere adquirir (acceder al dato “precio” del mismo modo que lo hicimos con “cantidad”).
- Una vez que comprobamos que cumple dichas condiciones, restamos 1 a las cantidad total del producto en cuestión.
- Utilizamos la f**unción “transfer” para enviarle al vendedor el precio del producto que han comprado**, es decir, e**stamos cobrando al emisor por la cantidad que ha comprado y se lo transferimos al vendedor**.
    - Para esto, tendremos que **utilizar la variable “vendedor” del struct**.
    - Realizar una **conversión explícita del address del “vendedor” a payable**.
    - Utilizar la **función transfer** pasándole como **parámetro el “precio” del producto**.
- Emitimos el **evento “ProductoComprado” pasándole todos los datos que requiere dicho evento**.

Paso 14. Implementar función obtenerProducto

Implementar la función "**obtenerProducto**" **pública que devolverá los datos de un producto específico. Recuerda establecer la mutabilidad de la función (pure, view o payable).**

***Parámetros de entrada**:*

- Variable **idProducto** de tipo **entero**.

***Parámetros de salida (recuerda que en el "returns" sólo debes establecer la tipología del datos, os dejo el nombre de manera representativa para que veais a qué se corresponde)**:*

- nombre (cadena)
- precio (entero)
- cantidad (entero)
- vendedor (address)

***Modificador de función**:*

- productoExistente(idProducto).

***Descripción**:*

- Accede al mapping por el idProducto que me han pasado en la función para devolver los datos del struct Producto correspondiente:
    - nombre
    - precio
    - cantidad
    - vendedor

(EXTRA - No realizar). Declarar función "receive"

Declaramos la **función receive() de tipo external y payable** sin cuerpo de función para que actúe de respaldo en caso de que alguien envíe ether desde fuera del contrato sin llamar a ninguna función.